# Java知识点

## 方法重载，何时用重载，为什么要使用重载？而不是把一个方法名字换成不同的。

举个例子，我们在一个商场买东西，要去收银台交钱，有三种支付方式：现金、刷卡、第三方支付，如果一种方式对应一个收银台，这就是把“支付”这个方法换成三个：现金支付(现金)、刷卡支付(信用卡)、第三方支付(手机)。重载就是无论用哪种支付方式在一个收银台都可以实现，就等于支付(现金)、支付(信用卡)、支付(手机)。

你觉得哪个比较好？

## 垃圾回收GC，什么时候会回收对象，怎么判断这个对象可以被回收。对象的的几种生存状态。

当对象不存活的时候，对象会被回收。判断对象是否存活，有两种方案，1是计数器，2是可达性。当计数器为0，表明没有别的对象引用该对象。不过计数器没法解决对象之间循环引用的问题。可达性分析就是通过GC Roots的对象作为起始点往下搜索，当GC Roots到某个对象不可达的时候，该对象就是可回收的。

## volatile

在多线程情况下，可以禁止指令重排。比如User user=new User()，这个类初始化步骤大概是：给user分配内存；调用User的构造函数初始化；将user的对象指向分配好的内存空间。指令重排简单说就是JVM一旦觉得某一步比较耗费时间（比如第二步），就先把第三步执行了。在单例模式的双重校验锁方法中，这样做就容易出问题，所以要用volatile关键字禁止指令重排。

## JVM类加载

这个真要详细说，一天时间都不够。简单说就是把java文件编译好的class文件，加载到内存里，对其进行校验、转换解析、初始化，最终形成可以被虚拟机直接使用的Java类型，这就是类加载机制。

类的生命周期：加载、连接（验证、准备、解析）、初始化、使用、卸载。

**加载：**

1. 通过类的全限定名获取定义此类的二进制字节流。
2. 将字节流所代表的静态存储结构转化为方法区的运行时数据结构。
3. 在内存中生成一个代表这个类的Class对象，作为方法区这个类的各种数据的访问入口。

**验证：**

1. 文件格式验证
2. 字节码验证
3. 符号引用验证

**准备：**

举例，public static int value=666;在准备阶段，value的值是0，到了初始化阶段才是666。如果public static **final** int value=666;则在准备阶段value的值就是666。

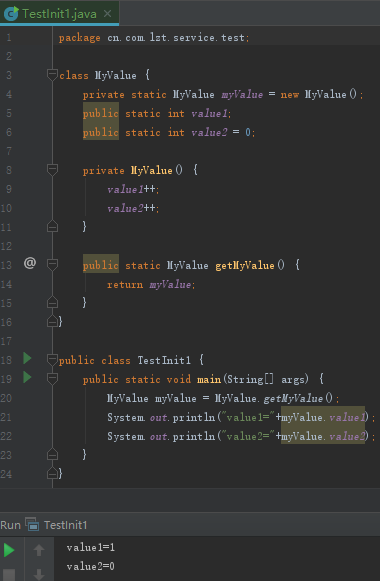
**初始化：**

在这个阶段主要执行类的构造方法。并且为静态变量赋值为初始值，执行静态块代码。

Java类只有对类的首次主动引用时才会初始化，主动引用的情况有：

1. new一个对象。
2. 访问或赋值类或接口的静态变量；调用类的静态方法。（静态变量或静态方法必须在这个类中，才会初始化这个类，如果在父类中定义，子类去调用，子类不会被初始化。另外final的静态变量如果是确定值的常量，所在类不初始化（不会执行static块）；如果是不确定值的变量，所在类需要初始化（执行static块））
3. 反射
4. 初始化子类时，如父类还没初始化，先初始化父类。（初始化一个类或接口时，父类必须先初始化，但其所实现的接口不会先初始化）
5. main方法所在的类。

例1：

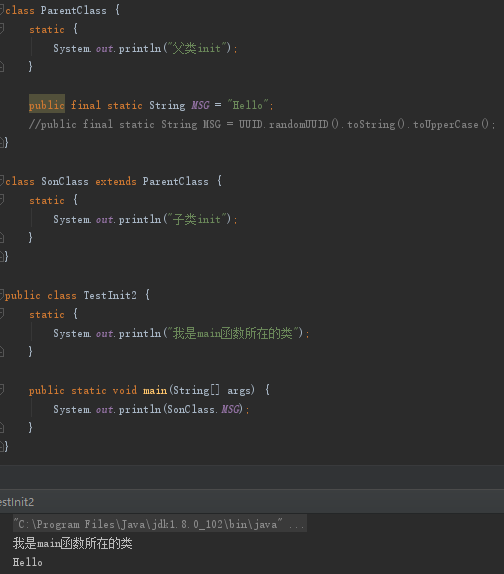


说一下整个的步骤：

1. 在main函数中，执行MyValue.getMyValue()。
2. MyValue类执行getMyValue()静态方法，需要初始化MyValue类（主动引用）。
3. 按照顺序执行new MyValue()，也就是MyValue的构造函数，value1和value2没有初始化的值都是0，++之后都是1。
4. 继续执行后面的两行，其中value1没有赋值，还是1；value2赋值了，由1变为0。
5. main函数打印value1、value2分别是1,0。

改一下，将new MyValue这一行挪到两个赋值语句的下面，执行结果就变成了1,1。原因就是先执行赋值，value1、value2都是0，然后再执行构造函数的++，就都变成1了。

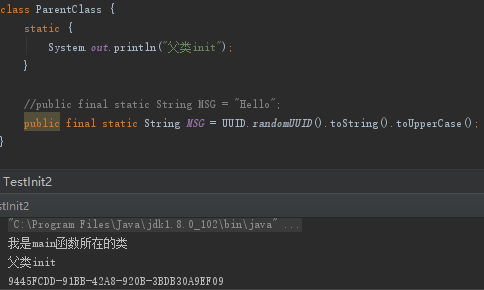
例2：



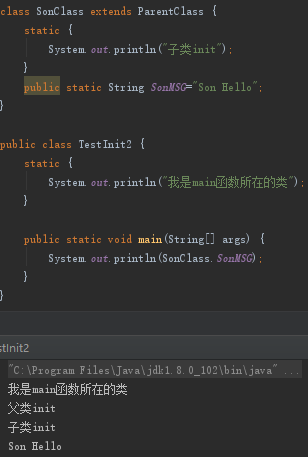
1. “我是main函数所在的类”肯定要先初始化，没啥可说的。
2. MSG这个静态变量是在父类中，所以子类不会初始化，也就不会打印出“子类init”。
3. MSG是final类型的静态变量，且是确定值的常量，所以ParentClass也不会初始化，不会打印出“父类init”。
4. 将MSG改为

public final static String MSG = UUID.randomUUID().toString().toUpperCase();

这个变量是不确定的变量，则ParentClass会被初始化，则“父类init”会被打印出来。



例3：



子类增加一个静态变量，引用的时候就需要初始化子类（注意，这里的静态变量不是final），按照初始化子类要先初始化父类的原则，先执行父类的初始化，打印出“父类init”，再初始化子类，打印出“子类init”，最后打印出子类的静态变量。

如果子类的静态变量改成final常量，

public final static String SonMSG="Son Hello";

则子类、父类都不会初始化。

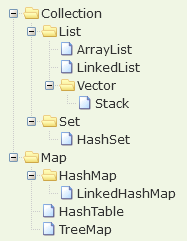
如果子类的静态变量变成final变量，

public final static String SonMSG=UUID.randomUUID().toString().toUpperCase();

则子类、父类都会初始化。

## Map,Collection（List、Set）

大概关系如下（包括但不限于）：



可自动扩展的数组——List

没有重复的数组——Set（最多有一个null元素）

自动排序的数组——TreeSet、TreeList等

如果说Collection存储的是一组对象，Map存储的就是键值对。

一个Map中不能包含相同的key，每个key只能映射一个value；null可以作为键，这样的键只有一个；可以有一个或多个键所对应的值为null。

**ArrayList和LinkedList：**

ArrayList基于数组，LinkedList基于双向链表，数组就没啥说的了，双向链表用图表示大概如下：



如果能理解这张图，那么ArrayList与LinkedList的区别也就能理解了。

ArrayList是数组，访问查询的时候是随机访问，而LinkedList则需要通过节点的指针一个个地移动，所以随机访问ArrayList占优势。

但是对节点的增删，LinkedList就占优，如上图我要在A与B节点中间插入一个节点C，就先生成一个节点C，然后把A指向后节点的指针从B改成C，把B指向前节点的指针改成C，C节点指向前节点指针指向A，指向后的指向B，就OK了。

ArrayList的增加元素与扩容请看下一节。

Vector可以理解成ArrayList的同步版，而Stack是Vector的子类，多了pop、push、peek、empty、search方法，可以使Vector成为一个堆栈。

**HashMap与HashTable：**

HashMap就是HashTable的非同步版，并且允许null key和null value。

LinkedHashMap是HashMap的子类，它维护了一个双重链接列表，定义了迭代顺序（插入或访问顺序）。

TreeMap是通过红黑树算法（要说清楚这个算法就太耗时间了，有空写一篇文章）实现的，取出来是排序后的键值对。

总结的说：如果需要按照自然顺序或者自定义的顺序遍历，用TreeMap。

如果要求输出顺序与输入顺序相同，用LinkedHashMap。

## ArrayList扩充问题。add()方法的底层实现

ArrayList增加元素相比LinkedList实现起来就麻烦，先抛开扩容不说,跟踪源代码我们可以看到，ArrayList增加或插入元素到最后是用到了System.arraycopy()这个方法进行自我复制，这过程是需要生成临时数组的，如果ArrayList比较大的话，这个耗时就可想而知了。

ArrayList的扩容，JDK7里面是用>>位运算，右移一位，容量扩大1.5。

JDK6则是通过copyOf（其实现也是arraycopy()）扩容1.5倍+1。

## Spring IOC原理&实例

没有IOC之前我们怎么用对象？new一个对吧。new完之后使用其方法，用完之后，这个类就没用了，等待GC。大家想一下我们会频繁地使用这个类，就需要频繁地去new，去使用，去GC，实际情况下使用的时间可能少，而创建、销毁消耗的时间更多，浪费。

所以我们有了单例模式，详情参看之前的文章。

而Spring IOC更进一步，由Spring来控制对象的生命周期以及对象间的关系。

举个例子，以前的方式就是我们自己做青椒肉丝，我们要自己去菜市场买肉买菜、处理肉菜、煎炒烹炸、吃饭、收拾锅碗。

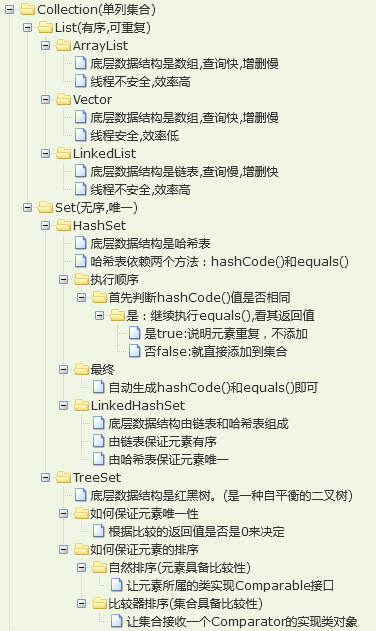
而Spring IOC就是去饭馆，直接说我要青椒肉丝，那么上面除了“吃”以外的环节都由饭馆来处理。方便很多了吧。

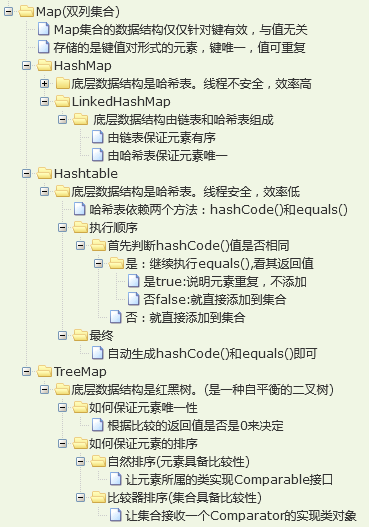
IOC默认实例化Bean都是单例模式（singleton），也有prototype——我们需要的时候，Spring会new一个给你。

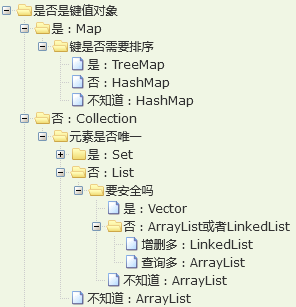
什么时候用呢，举个例子，有两个类，User（存储用户信息）和LoginCheck（校验用户信息），LoginCheck就应该是单例，因为它无状态，不管谁，传什么样的用户信息，我校验就是了，所以有一个实例就够了。

但是User就不能是单例了，你我他的用户信息能一样么，所以用一次new一次才是对的。

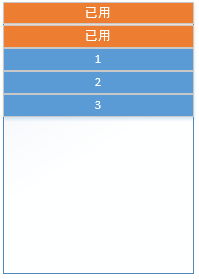
## Collection与Map的一些知识点图示



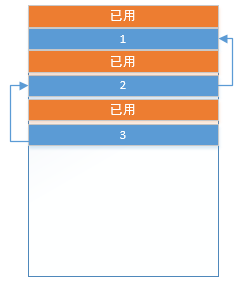




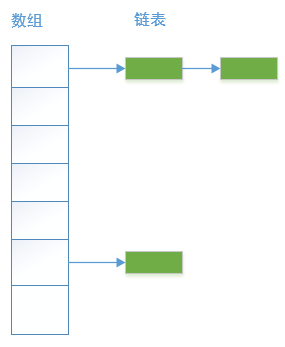
在内存中数组是连续存储：



链表则是离散存储：



HashMap则是数组＋链表：



数组的每一个元素都是链表。

## Minor GC和Full GC的触发时机

说几个基本概念：

HotSpot的采用分代收集算法（新生代，老年代）。采用可达性分析法判断对象是否还存在引用，从GC Roots对象为起始点，往下搜索，如果某个对象到GC Roots没有任何引用链相连，则此对象是可回收的。但是也需要两次标记：

1. 对象没必要执行finalize()方法（对象没覆盖finalize()；finali()已经被虚拟机调用过），直接回收。
2. 对象有必要执行finalize()方法，放在F-Queue队列中，稍后会由虚拟机创建的一个优先级别低的线程去执行，进行第二次标记。

什么时候GC：调用System.gc时；系统自动GC时：当Eden区和From Survivor区的内存不够时，启动GC。

如何GC：通过可达性分析，将Eden和From Survivor区存活的对象复制到To Survivor区，然后释放这两个区，当对象在Survivor区熬过15（默认）次Minor GC，就进入老年代。

**Minor GC触发条件：**当Eden区满时，触发Minor GC。

**Full GC触发条件：**

（1）调用System.gc时，系统建议执行Full GC，但是不必然执行。

（2）老年代空间不足。

（3）方法去空间不足。

（4）通过Minor GC后进入老年代的平均大小大于老年代的可用内存，主要是避免新生代对象晋升老年代时，老年代空间不足的情况。

（5）由Eden区、From Space区向To Space区复制时，对象大小大于To Space可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小。

## 反射获取private成员和方法



关键在setAccessible(true)这个方法。

## 反射机制用到哪些类

Class：类对象

Constructor：类的构造器

Field：类的属性

Method：类的方法

## 阻塞非阻塞&同步异步

举个例子：

1、你吃麦当劳，在柜台点餐之后等待，直到拿到食品。

2、你吃麦当劳，在柜台点餐之后不等，出去逛了一圈之后去柜台问问，要是食品做好了，拿走吃；没做好就再出去逛一圈。

3、你吃麦当劳，在自助点单机点单之后，拿到号，在店内等待叫号。

4、你吃麦当劳，拿起手机叫了个外卖。

同步异步，指的是调用方与被调用方之间的**通信机制**。第一、二种情况，你想吃饭，要自己去获取这个结果（死等或主动询问），这是同步。第三、四种情况，你点餐之后不需要等结果，工作人员通知你，这是异步。

阻塞非阻塞，指的是调用方在**获得结果这段时间的状态**。第一、三种情况，你只能在店内等，干不了其他事，这就是阻塞。第二、四种情况，在得到结果这段时间，你是自由的，可以去做其他的事，这是非阻塞。

第一种情况就是同步阻塞（BIO），第二种情况是同步非阻塞（NIO），第三种情况是异步阻塞，第四种情况是异步非阻塞（AIO）。

## NIO相关知识点

NIO相比传统BIO多了以下几个概念：

Channel，通道。传统IO的流（inputstream、outputstream）是单向的同步的，通道则是双向的异步的。它与Buffer进行交互。

Buffer，缓冲区。就是一块存数据的内存。

Selector，选择器。相当于一个观察者，用来监听通道感兴趣的事件。

去麦当劳用过自助点单机的读者应该注意到，会有个负责叫号的员工，她会关注后厨有没有把所有单号的餐都做好，一旦你（Channel）的单号（SelectionKey）里面的餐都做好了（放入Buffer），她就会去收集这些餐点放入餐盘，然后叫号通知你。

注意！她来通知你，这是异步，就变成AIO了，而NIO中的，她只是负责把餐收集齐了，放在一个餐盘里（就绪状态），是你不停地根据单号去主动柜台检查“2673号的餐好了没？”，你一看就绪了，拿走吃，这是同步。

不管是AIO还是NIO，不管你是等叫号，还是主动去问，在拿到餐之前，都是非阻塞的。

NIO是Reactor模式，AIO是Proactor模式。

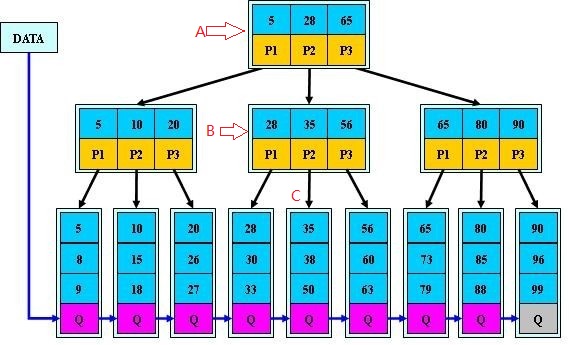
## B+树&索引

说B+树之前，要先提一下磁盘IO与预读，众所周知，机械磁盘IO一次成本是很高的，为了解决这个问题，操作系统做了一些优化。这又要说到局部性原理，尤其是空间局部性：对于最近将来要用到的信息，很可能与这次使用的信息在空间地址上是临近的。

打个比方，有六个学生：刘大、马二、张三一同坐在教室的最前排；李四、王五、赵六坐在教室的最后排。班主任想安排人去打扫卫生，通常的做法是选择这两组人中一组（空间临近），而不是从前排挑1个，再从后排挑2个。

根据这个原理，操作系统的预读的时候，不仅把当前磁盘地址的数据，还把相邻的数据一起读取到内存里。

一个3阶B+树查找过程：



比如我们查找38。

1. 把磁盘块A读进内存，二分法查找，28<38<65，定位P2指针。
2. 通过P2指针找到磁盘块B，读进内存，35<38<56，定位指针P2。
3. 通过P2指针找到磁盘块C，读进内存，找到38。

一共就3次磁盘IO，节省资源。

以上可得出一些建立索引的原则：

1. IO次数=B+树高度。

一个磁盘块的大小是固定的，由操作系统决定，比如4K。那么如果我们设置的数据项越小，在一个磁盘块中存的就越多，这样一来，同样的数据量，高度就越小。

所以很多数据库优化教程中，MySQL主键设置爱用自增的int，Oracle爱用序列Sequence，而不建议用字符串的UUID，毕竟数值所占字节少嘛。

1. 数据量上的字段没必要加索引。这句话的意思是说，如果一个字段里的数据就只有了了几种内容，就不要给这个字段加索引了，比如说性别或状态，记录里面的内容都是0,1，加了索引，在B+树的磁盘块A里，放上0,1两个数，剩下的值分布概率50%，要知道，在MySQL中，请求表的数据行数超过总记录的30%，就会变成全表扫描，所以加这个索引没意义。加了索引每次新建删除更新还得多操作一次索引，得不偿失。

除非这个字段经常被查询，并且数据总量小于30%，比如订单状态字段，0代表为完成，1代表已完成，通常情况下，未完成的订单少，经常查，这时候倒是可以加上索引。

1. 最左前缀匹配原则，举个例子，有表A，字段A、B、C，记录如下：

|  |  |  |  |
| --- | --- | --- | --- |
| rowid | A | B | C |
| 1 | 1 | 1 | 6 |
| 2 | 1 | 1 | 7 |
| 3 | 2 | 1 | 3 |
| 4 | 2 | 1 | 4 |
| 5 | 2 | 2 | 1 |
| 6 | 2 | 2 | 2 |
| 7 | 1 | 2 | 8 |
| 8 | 1 | 2 | 9 |

我们设置复合索引为a，b，其实就等于创建了两个索引，a和（a，b），那么这个复合索引就应该如下：

|  |  |  |
| --- | --- | --- |
| A | B | rowid |
| 1 | 1 | 1 |
| 2 |
| 1 | 2 | 7 |
| 8 |
| 2 | 1 | 3 |
| 4 |
| 2 | 2 | 5 |
| 6 |

当你查询where a=2 and b=1 时，通过二分法，先根据a=2定位到3、4行，再根据b=1，定位到第三行里的两个rowid。但是如果你只where b=1，没有这个B+树索引，所以只能全表扫描。

1. 索引列不能参与函数计算，这个应该很好理解吧，举一个我刚参加工作时犯的错误。一张大表，要查询时间，我对这个字段加了索引，数据库里的时间字段其实存的是时间戳嘛，而我们传过来的基本是字符串“2017-08-29”这种。

我当时的做法是TO\_CHAR(时间字段)=“2017-08-29”，这是先把时间戳字格式化为字符串，那肯定就不用索引了，必然是全表扫描，所以改成 时间字段=TO\_DATE(“2017-08-29”)就可以了，把传入的字符串转成时间戳，这时候就用上索引了。

## JVM运行数据区

### 程序计数器

当前线程执行的字节码行号指示器。字节码解释器通过改变它的值来选取下一条要执行的字节码指令。一个线程对应一个程序计数器。

执行Java方法，里面存的是正在执行的虚拟机字节码指令的地址；执行Native方法，为Undefined。

### 虚拟机栈

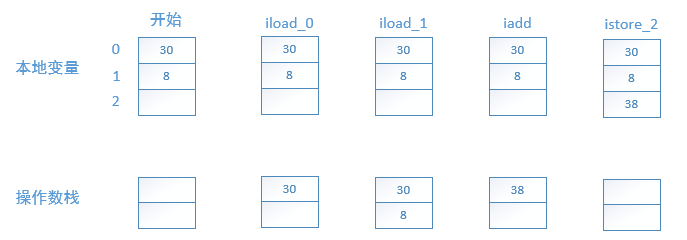
与线程的生命周期相同。每个方法被执行的时候会同事创建一个栈帧，用于存储局部变量表、操作数栈、动态链接、方法出口等信息。

#### 局部变量表

存放了编译期可知的各种基本数据类型、对象引用（reference）、returnAddress类型（指向一条字节码指令的地址）。

#### 操作数栈

是一个以字长为单位的数组，通过栈操作来访问。大多数操作要从这里弹出数据，执行运算，然后把结果压回操作数栈。



#### 动态连接

编译时，会得到程序中每个类或接口独立的class文件。而这些类看起来独立，但又通过符号引用相互关联。这些符号引用放在常量池里。如果A想调用B的方法，虚拟机的方法调用指令就用B的符号引用作为参数，而符号引用不是直接指向B的内存地址，所以需要转换成直接引用，才能访问到方法。

如果符号引用在类加载阶段或第一次使用时转化为直接引用，就是静态解析。如果是运行期间转为直接饮用，就是动态连结。

#### 返回地址

有两种，一种是正常退出，根据方法的定义来决定是否要把结果返回给调用者，把返回值压入调用者的操作数栈中，并且把程序计数器的值改为调用入口的下一条指令。

一种是异常，不会有返回值给调用者。

#### 异常

如果线程请求的栈深度大于虚拟机所允许的深度，将抛出StackOverflowError 异常。

虚拟机栈可以动态扩展的时候，当扩展时无法申请到足够的内存时会抛出OutOfMemoryError 异常。

### 本地方法栈

与虚拟机栈类似，虚拟机栈执行Java方法，本地方法栈是为使用的Native方法服务。有的虚拟机直接把二者合二为一。

### 堆

存放实例对象，所有线程共享。

因为采用分代收集，堆中分为：

新生代：程序新建对象都是现在新生代分配内存。由Eden区、From Survivor区、To Survivor区组成。

老年代：在新生代经过多次（默认15）Minor GC存活下来的对象，以及大对象、大的数组对象。

要是在堆中没有完成实例分配，堆也扩展不了的时候，报OutOfMemoryError异常。

### 方法区

存储了每个类的信息（包括类的名称、方法信息、字段信息）、静态变量、常量以及编译器编译后的代码等。

在JVM的规范中，没有强制要求方法区必须实现垃圾回收，所以又被称为“永久代”。

#### 常量池

常量池在方法区内，用来存储编译期间生成的字面量和符号引用。在类和接口被加载到JVM后，对应的运行时常量池就被创建出来。

## JVM内存分配流程

1. 为实例的对象在Eden区分配内存。
2. Eden区空间不够，触发Minor GC。
3. 如果空间还不够，将部分对象放入Survivor区。
4. 如果老年代空间足够，新生代对象熬过一定次的Minor GC，就会挪到老年代。
5. 如果老年代空间不够，触发Full GC。
6. Full GC之后，Survivor和老年代还是没空间存放Eden复制过来的对象，JVM就没法在Eden区给新对象分配内存，报OutOfMemoryError异常。

说一下为什么要有两个Survivor区。Survivor区主要作用是在Eden区和老年代做一个缓冲，如果没有Survivor区，每个Eden区的对象，Minor GC一次就往老年代存，很快就把老年代填满，就得触发Full GC——这个操作很耗费资源，消耗时间的。

用两个Survivor区主要是因为现在流行的虚拟机采用复制算法，先用一块区，满了之后把存活的对象复制到另一个区，不存在内存碎片问题。

## 垃圾收集算法

现在流行的虚拟机用的都是分代收集算法，根据新生代、老年代的特性，使用不同的垃圾收集算法。

### 标记-清除

回收前：



回收后：



这有一个问题——内存碎片，假设马上要分配4块内存给一个新的对象，上面的内存区搞不定了。

### 复制

回收前：



回收后：



虽然内存比前面的少了一半，但是解决了内存碎片的问题。而且由于Eden区的对象98%是朝生夕死，所以Eden与Survivor区分配的比例是8：1。

这种算法用在新生代。

### 标记-整理

回收前：



回收后：



老年代用。

## 接口与抽象类

个人的理解，假设有一个抽象类：人类，有一个私有的方法：生孩子()。有一个抽象类：卵生动物，有一个私有的方法：下蛋()。有一个接口：动作行为，有一个公共方法：吃()。

那么一个我们生成类应该是：

女人 继承 人类 实现 动作行为。

母鸡 继承 卵生动物 实现 动作行为。

以上两个的继承是不能乱的，但是吃这个动作行为是都有的。

抽象类决定了你这个类是什么，接口决定了你这个类要做什么。

## HTTP请求到响应的流程

1. 域名解析。
2. TCP三次握手。

2.1、C端发送syn（同步序列号）=i到S端，进入SYN\_SENT状态。

2.2、S端收到syn包，确认SYN报文，ack=i+1；并发送自己的syn=j，进入SYN\_RECV状态。

2.3、C端收到ack+syn，向S端发送确认报文ack=j+1。C和S端进入ESTABLISHED连接成功状态。

1. S端解析HTTP请求。

HTTP请求有四部分：

3.1、请求行：方法、URI路径、HTTP协议版本号。

3.2、请求头：包含缓存相关信息、客户端身份信息。

3.3、空行。

3.4、消息体：C端发给S端的请求数据。

1. S端响应请求，返回数据。

HTTP响应有四部分：

4.1、状态行：HTTP协议版本号、状态码、状态说明。

4.2、响应头：回传给客户端说明服务端相关情况的信息。

4.3、空行。

4.4、响应体：S端返回给C端的响应数据。

1. 浏览器解析。
2. TCP断开连接。

断开连接的操作可以是C端或S端任意一端发起：

6.1、A向B发送FIN=i（finish）报文，进入FIN\_WAIT\_1状态，告诉B我没有数据要发给你了。

6.2、B向A返回ACK=i，进入CLOSE\_WAIT状态，告诉A我知道你已经没有数据要发给我了，但是我可能还有数据要发给你，你先等等。

6.3、B向A发送FIN=j，进入LAST\_ACK状态，B彻底没数据要发了，告诉A，我要关了。

6.4、A进入TIME\_WAIT状态，返回ACK=j，B进入CLOSED状态，A告诉B，我知道了，你关吧。A在等待2个MSL（Max Segment Lifetime 报文最大生存时间）之后真正关闭。

为什么断开要4次握手，因为TCP有个半关闭状态，即A没有数据，要关闭了，但是B还有数据发给A，所以要等B数据都发完了，B再通知A自己要关闭。

## Spring事务

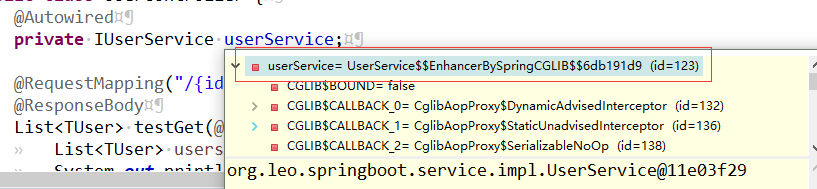
先看一下JDBC事务的流程：

1. 获取连接Connectio conn = (Connection) DriverManager.getConnection(url, username, password);
2. 设置事务提交方式为手动 conn.setAutoCommit(false);
3. 执行SQL语句
4. 正常则提交事务/异常则回滚事务 conn.commit()/conn.rollback();
5. 关闭连接 conn.close();

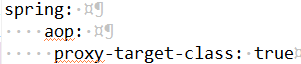
因为我用的是MyBatis和Druid连接池，所以与Connection相关操作由这两个来负责。

而事务方面，即2、4步就由Spring来负责。

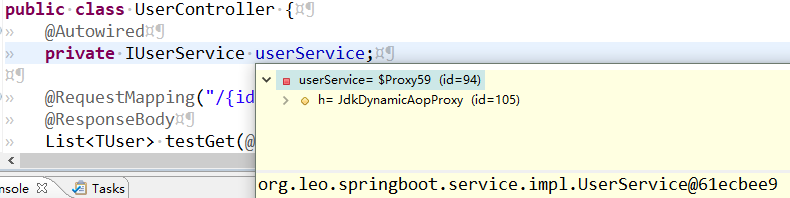
Spring AOP采用了代理，所以我们在Controller注入的UserService其实是代理类，具体可见：



使用的是CGLIB，因为我在application.yml中做了如下配置：

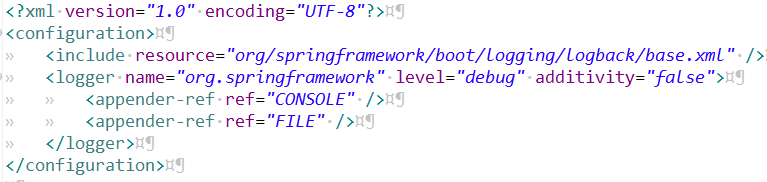


如果把true改成false，就会用JDK代理，如下：



为了更方便的看清Spring执行事务的过程，我将LOG日志打印出来。

在resources目录下创建logback-spring.xml，内容如下：



执行一次请求，列出如下日志：

|  |
| --- |
| o.s.j.d.DataSourceTransactionManager : **Creating new transaction** with name [org.leo.springboot.service.impl.UserService.insert]: PROPAGATION\_REQUIRED,ISOLATION\_DEFAULT; ''  o.s.j.d.DataSourceTransactionManager : Acquired Connection [com.mysql.jdbc.JDBC4Connection@c76cf3b] for JDBC transaction  o.s.j.d.DataSourceTransactionManager : Switching JDBC Connection [com.mysql.jdbc.JDBC4Connection@c76cf3b] to manual commit  **//这里在执行SQL语句**  o.s.j.d.DataSourceTransactionManager : **Initiating transaction commit**  o.s.j.d.DataSourceTransactionManager : **Committing JDBC transaction on Connection** [com.mysql.jdbc.JDBC4Connection@c76cf3b]  o.s.j.d.DataSourceTransactionManager : Releasing JDBC Connection [com.mysql.jdbc.JDBC4Connection@c76cf3b] after transaction  o.s.jdbc.datasource.DataSourceUtils : Returning JDBC Connection to DataSource |

可以清晰地看到Spring通过AOP，使用DataSourceTransaManager对事务的处理过程。

## 悲观锁&乐观锁

面试的时候遇到过问Redis是如何解决“竞态条件”的，相关知识点总结一下。

所谓竞态条件，举个例子，一个代表点击数的数值hitcount，每个客户点击一次则＋1。

没有事务的时候，假设我们的操作如下：

|  |
| --- |
| hc=GET hitcount;  hc=hc+1;  SET hitcount $hc; |

非并发状态下，这样做是OK的，但是并发状态下会出现的问题是：



A和B两个客户端分别从Redis处取值，并+1，值都是11。



Redis是单线程模型，所以A和B的SET命令只能先执行1个，此处先执行A，hitcount更新为11。



接着执行B的SET命令，hitcount依然是11，这就是明显的因为竞态而产生的错误，hitcount应该为12才是。

Redis的事务其实是通过MULTI命令开启事务，将后续一系列的命令放在一个队列里，不立即执行，直到EXEC命令，队列中的命令才会依次执行。

命令类似：

|  |
| --- |
| MULTI;  set val1 111;  set val2 222;  EXEC; |

在实际工作中，我们也会经常遇到这种问题：我们必须先拿到数据，根据数据做出判断，进行一些处理之后，再更新数据，这时候我们就没法保证取数据，更新数据在同一个队列事务中了。

这就需要乐观锁。简单说，我们每取一个数据的时候，Redis不仅返回数值，还会返回这个数值的版本号。当我们执行更新命令时，Redis会拿你要SET值的版本号与库里现在值的版本号进行比对，如果相同，则更新，版本号变更。如果版本号不同，则说明在我们执行更新命令之前，有其他客户端修改了这条数据，我们的更新操作失败。

Redis里是通过WATCH命令来监控版本号的。

|  |
| --- |
| WATCH hitcount;  hc=GET hitcount;  hc=hc+1;  MULTI;  SET hitcount $hc;  EXEC; |

因为通过WATCH监控了hitcount这个Key，那么在事务中SET的时候，一旦发现版本号不对，执行就失败。

**悲观锁：**

乐观锁是CAS——Check And Set，先检查（版本号）再设置更新，那么悲观锁就是先锁，再查，最后更新。

以MySQL为例，我们要实现悲观锁：

SELECT \* FROM tabA AS t WHERE t.id=1 **FOR UPDATE**;

这样，因为主键是明确指定的（id=1），所以这一行记录就被锁定了——**行锁定**，在本事务被commit之前，这条数据都只会被本事务的SQL语句进行修改，其他事务的更新操作都会在本事务提交之后再执行，单纯的查询则不受影响。

如果没有指定明确地主键，则锁定的是表——**表锁定**。

## 幂等性

### 前言

在前一篇文章《Spring Cloud Feign重试》中，谈到了接口的幂等性问题，在这里结合我实际工作中的情况谈谈处理的思路。

解决思路没有完美的，总是有优点优缺点，所以实际工作中还是要结合实际来选择。

### 重试&幂等性

在一个系统中，无论是对外还是内部的接口API，遇到异常失败的场景还是很常见的，网络抽风了，内存溢出了，硬盘坏道了，硬件的问题软件的问题都会导致某个接口调用失败，作为调用方，为了用户体验也好，系统的健壮性也好，要有容错机制，比如failover、failfast。

其中failover就会设置接口调用的重试次数，failfast则直接报错不重试。

一旦重试，就涉及到被调用接口的幂等性问题。

如果接口时查询操作，那么重试多少次都OK，顶多耗点资源，对数据不造成影响。

但如果是增删改这些写操作，重试就比较麻烦了，就像我在《Spring Cloud Feign重试》一文中举的例子，接口向数据库插入一条数据成功了，也返回了，但是因为网络抽风等原因，被调用方没有及时收到响应，导致超时异常，如果被调用方采用的容错机制是failover，进行重试，那么被调用的接口又会重新增加一条数据。这就很糟糕了。

### 解决方案——调用方

结合我实际工作中应用的Dubbo、Spring Boot来说说。

Dubbo是可以针对被调用方设置容错机制的，所以完全可以把幂等性操作的接口设置为failover，并设置retry的次数。

非幂等性的写操作接口，设置为failover。

Spring Cloud，通过Feign调用接口的时候，针对写操作就不必配置重试，Feign的重试默认是不开启的。查询操作再开启重试。

### 解决方案——被调用方

如果是一位后端接口开发人员，我们是不能指望调用方会考虑幂等性问题的。所以这个事还得是我们自己来。

#### **全局唯一ID**

作为被调用方，在非幂等性的接口协议上，要求调用方在传递业务数据的同时带上一个ID，形式可以多种多样，比如UUID、随机数、被调用方ID+时间戳等，保证全局唯一即可。

在接口执行具体业务逻辑之前，通过网关（比如Spring Cloud的Zuul）、Spring的AOP，拿到ID，去某个数据库查询（比如Redis、Memcached、MySQL等），如果存在则说明调用过了，直接返回。如果不存在，则说明这次调用是第一次，通过AOP等方式，将ID存入数据库（强烈建议用Redis等内存数据库），以便后续查验。

当然，这个存入的ID要设置一定的失效时间。

#### **数据库解决方案**

数据终究要持久化。有几个解决方案

**新增数据**。我们可以设置某个字段唯一索引，比如主键。由被调用方设置，这样一旦重试新增操作，因为唯一约束必报异常，事务就会回滚。

**SaveOrUpdate**。MySQL里有个“INSERT....ON DUPLICATE KEY UPDATE”语法，意指如果唯一索引的字段有重复，则执行更新，否则新增。

**更新操作，版本号**。这个其实就是某种乐观锁的实现了，我们可以在记录上指定一个字段做版本号，比如更新时间戳，独立的版本号字段。在执行update语句的时候，WHERE语句应该类似：

|  |
| --- |
| updat tableA set name=#{name},**upd\_time=#{newUpdTime}** where id=#{id} and **upd\_time=#{oldUpdTime}**  或  updat tableA set name=#{name},**version\_num=#{newVerNum}** where id=#{id} and **version\_num=#{oldVerNum}** |

## MyBatis#与$的区别

就是类似JDBC里预编译SQL语句与拼接SQL语句的区别。

|  |
| --- |
| select \* from user where id=#{id}; |

相当于JDBC里

|  |
| --- |
| String SQL="select \* from tuser where id = ?";  PreparedStatement ps = conn.prepareStatement(sql);  ps.setInt(1,这里应该放id具体的值，或者变量名id); |

$就相当于拼接字符串

|  |
| --- |
| select \* from tuser order by ${id}; |

等于JDBC里

|  |
| --- |
| String strOrderByField=”id”;  String SQL=”select \* from tuser order by ”+ strOrderByField; |

最好肯定是用#，防止SQL注入。

## SQL语句执行顺序

|  |
| --- |
| SELECT 要查询的列 DISTINCT 要去重的列 FROM 表1 LEFT/RIGHT JOIN 表2 ON 条件 WHERE 查询条件 GROUP BY 分组列 WITH ROLLUP/CUBE HAVING 条件 ORDER BY 排序列 LIMIT 指定行 |

这基本上是一个很全的SQL语句了，执行的顺序：

1. FROM。FROM左表和右表进行笛卡尔积计算，产生虚拟表VT1。
2. ON。通过ON后面的条件，对VT1进行筛选，产生VT2。
3. JOIN。将保留表未匹配的记录添加到VT2中，产生VT3。
4. WHERE。对VT3进行过滤，产生VT4。
5. GROUP BY。对VT4进行分组操作，产生VT5。
6. ROLLUP/CUBE。对VT4进行ROLLUP/CUBE操作，产生VT6。
7. HAVING。对VT6进行HAVING过滤，产生VT7。多说一下HAVING，与WHERE一样都是通过条件过滤结果，但是HAVING过滤的是组，WHERE是行；WHERE后不能用聚合函数（SUM/COUNT/MAX等），HAVING可以。
8. SELECT。选择指定的列，产生VT8。
9. DISTINCT。对VT8去重，产生VT9。
10. ORDER BY。对VT9排序，产生VT10。
11. LIMIT。在VT10中取出指定的记录，产生VT11。

## 观察者、反应器、前摄器

前段时间去面试，被问到Dubbo的底层实现。其缺省使用Netty，是一个NIO异步通信的单一长连接。很适用于消费者多与提供者。

再深一点就会问到Netty为什么能做到高性能，答案是实现了Reactor（反应器）模式。

这里就又引申出Proactor（前摄器）模式，而且很容易跟Observe（观察者）模式——也叫发布/订阅模式混。这里结合实际工作，讲讲我对这三个的理解。

先说观察者模式，可以先看《Redis消息队列基于发布订阅模式》，结合代码看应该会比较清楚。

假设，我们的程序有这么几个功能块：发帖，回帖，记日志，加积分，推送。

假设我们这么设定流程：

发帖之后要记日志、加积分、推送给相关人；

回帖之后要记日志、加积分。

如果我们采用消息队列来处理，其实就可以可以创建两个Channel：post、reply。

其中记日志、加积分可以订阅post、reply，推送只订阅post。

当发帖或回帖动作完成之后，分别向post、reply发送消息，后续的功能模块就能接到通知进行下一部分的操作。

而以后如果流程变了，回帖之后不加积分了，又多了推送功能，那相应的功能块进行解除订阅、订阅操作即可。

Reactor和Proactor与之不同的是，他俩是针对的是I/O。

简单总结就是，Reactor：邮局啊，如果有我的信了，通知我一下，**我去拿**。Proactor：快递啊，我的信给我送到家了（**我不去拿**），通知我一下，我来处理。

二者都是要先注册，但是

Reactor是同步I/O——能接收数据了，调用我去接收并处理。

Proactor是异步I/O——已经接收完数据了，调用我去处理。

这里就容易把观察者与Reactor搞混，因为都是订阅——发生改变——通知。但是Observe是与单个事件源关联：多个观察者去监控一个Channel。而Reactor是与多个事件源关联，不停地进行selector轮询，一旦有注册的事件发生，通知具体的对象去处理。因为复用了线程，所以避免了线程切换同步数据移动带来的性能问题，所以适合处理时间短、数据量小的场景，Dubbo的默认协议适应的场景，就是基于此来的。

## 间隙锁

说间隙锁之前要说一下**幻读**。

举例：

事务A：SELECT count(id) FROM tabA AS t WHERE id>100;

结果为10。

事务B：INSERT INTO tabA(id,name) VALUES(101,’Leo’);COMMIT;

事务A：SELECT count(id) FROM tabA AS t WHERE id>100;

结果为11。

这就是幻读。

额外说一句，都是第一次查询的结果与第二次查询的结果不一致，不可重复读重点在数据的修改，幻读的重点在新增或删除。

首先说明，MySQL的默认隔离级别是“可重复读”，在这个级别下是避免不了“幻读”的。为了解决此问题，采用的是MVCC（多版本并发控制技术）。这个我回头再开一篇文章写，大家先记住在MVCC下，读取分为两种“快照读”、“当前读”，间隙锁就是用在“当前读”，来防止出现幻读。下面介绍一下机制。

之前介绍悲观锁和乐观锁的文章里有写到加悲观锁的方法：

SELECT \* FROM tabA AS t WHERE t.id=1 **FOR UPDATE**;

因为指定了主键，所以只有id=1的那一行记录被锁定。

如果是这种写法（假设记录ID是自增，分别是1、2、3……99、100）：

SELECT \* FROM tabA AS t WHERE t.id>99 **FOR UPDATE**;

查询的是一个范围，是“99”到“无穷大值”中间的“间隙”，把这个“间隙”锁住，就是“间隙锁”。

这样做的好处，按照文章开头的例子，事务A不仅把表中已经有的记录（id=100）给锁住，还把之后没有的记录（id=101、102、103……）也锁住了。这样事务B想插入id=101的记录就需要等事务A结束，从而避免了幻读。

## 缓存穿透、缓存雪崩

### 缓存穿透

客户端请求了一个不存在的Key，在缓存中获取不到，只好不停地从DB查询，从而导致DB挂掉。

解决方案：

1. 构建一个大大的bitmap，里面存着可能存在的Key。每次从缓存取数据之前，过滤一下Key，如果不在其中，就不用向DB发请求了。
2. 将不存在的Key设一个NULL值，避免再向DB发请求。这个Key的失效时间要短一些。

### 缓存雪崩/击穿

雪崩是N多Key，击穿是1个Key。

如果某些（个）热点Key，失效了，而这时有高并发过来访问，会导致后端的DB、接口等压力大增。

解决方案：

1. 永久不过期。
2. 针对某些热点Key，设置的不同的随机的过期时间，保证这些Key不在某个集中时间点上一起过期。
3. 第一个请求获取Key失效，进行重新缓存操作的时候，给资源加锁（单机用同步锁，分布式用Zookeeper的分布式锁），保证后面的请求等待缓存重建完成。

## 强、软、弱、虚引用

之前的文章谈了很多JVM的垃圾回收机制、算法等，而看一个对象是否会被回收，主要看是否有引用指向该对象。

Java有强、软、弱、虚引用。

其目的是：1、利于垃圾回收；2、便于程序员通过编码来决定对象的回收周期。

1. 强引用

最常见的的，A a=new A()，只要强引用在，JVM宁可报OOM，也不会回收。

可通过a=null的方式，使JVM回收此对象。

1. 软引用

内存空间够，就不回收。

在发生OOM之前，JVM会尽可能地回收长时间不使用的软引用对象。

1. 弱引用

无论内存空间是否充足，都会被回收。

1. 虚引用

与没有引用与之关联一样，任何时候都会被回收。

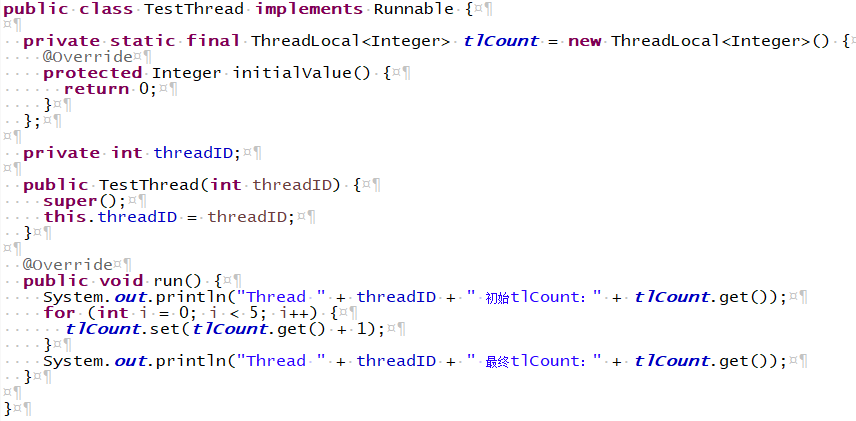
软引用可用在缓存上。比如加载大量的文件或图片。如果每用一次就加载一次，会受到硬盘、网络等的限制，影响性能。可如果全加载到内存，又会导致内存不够用，而溢出。

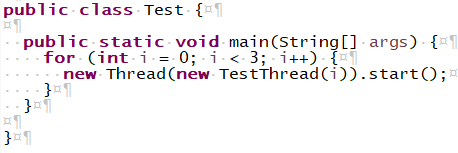
软引用就可以很好地解决此问题。将文件图片的缓存对象设置为软引用，当内存足够的时候，程序可以访问缓存，避免读取硬盘影响性能；当内存不够的时候，JVM回收这些缓存对象，避免OOM。

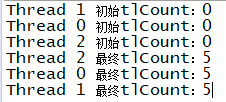
## ThreadLocal

只要涉及到多线程，就必然要考虑变量的线程安全。有一种全局变量，多个线程都会对其进行操作，为其安全性我们可以采取synchronized、volatile等方式来处理。

还有一种局部变量，它只存活于线程内部，供线程内的函数使用，与其他线程无关，这就要用ThreadLocal。







各个线程内的ThreadLocal变量是相互独立的，互不干扰。

### 实现思路1

在每一个ThreadLocal类中构建一个Map，以线程ID为Key，具体的值为Value。如图：



JDK早期的实现是如此。

### 实现思路2

每个Thread维护一个ThreadLocalMap，ThreadLocal为Key，具体的值为Value。如图：



这样做好处有：

1. 原来要维护一个大Map，有多少个线程，Map里就要有多少个键值对。比如我们有3个ThreadLocal变量，10个线程，如果要获取数据，就要在一个有10个键值对的Map里查找。

现在则是在10个线程里，维护一个只有3个键值对的Map，查找就快了。

1. Thread运行完毕，销毁后，内部的ThreadLocalMap也会一起销毁，节省内存。